Automating the Parallelization of Functional Programs

Michael Dever and G. W. Hamilton
{mdever, hamilton}@computing.dcu.ie

Dublin City University

Abstract. Creating efficient parallel software can be a complicated and time consuming task, as there are many issues that need to be considered, such as race-conditions, thread-bound data dependencies and locking issues, among others. An automated parallelization system could solve such issues, and would be of huge benefit to developers. Such a system would ideally take in a sequential program and produce, using program transformation, an optimized, parallel and equivalent program, without human interaction. There are two main approaches to the transformation of functional programs: fold/unfold based systems, based on the works of Burstall and Darlington [7], and calculational methods based systems, based on the Bird-Meertens Formalisms [3, 2, 31, 14, 1, 37]. In this paper we examine existing works on automating the parallelization process, specifically that of functional languages and review and compare their contributions to the field.

1 Introduction

Parallel Programming is the process by which developers take a problem and break it into sub-problems that can then be executed in parallel, hopefully reducing the amount of time it takes for the problem to be solved. There are many known problems with a parallel approach to programming, such as race-conditions, problems working with data shared between parallel code, locking problems such as deadlock, and many others. These all have to be taken into account by the developer, and this can be an arduous task.

Implementing parallel software by hand can require a great deal of time, knowledge and skill. In the real world, at least one of these requirements can often be missing, and in an industrial environment, time is usually a major constraint on development. As an example, a developer implementing a program in parallel may have to deal with non thread-safe operations on the data the developer is using. This would mean that the developer has to consider what would happen if the data changes mid-thread and what would happen if this change occurred on data shared amongst several threads, among other problems.

There are many different approaches to parallelism in computing, such as task parallelism, a software based approach, in which each processor executes a potentially independent block of code on potentially independent sets of data. Data parallelism [23] is another software based approach, in which different processors perform the exact same instruction on independent nodes. This survey focuses upon the area of data parallelism, in which data is distributed across nodes (a core on a processor can be a node) for concurrent processing [12].

Skeletons, due to Cole [11], are another approach that can be used to aid in the creation of parallel programs. These are parallel primitives, or functions, that can be seen as building blocks to be used in the creation of a parallel program and that exist at the core of the programs parallel implementation. The main benefit of using skeletons is that they have well known parallel implementations, and as such remove the need for the developer to worry about this.

In order to automate the parallelization process, a technique known as *program transformation* can be employed. Program transformation describes the process of taking an input program and transforming it via various methodologies, into a semantically equivalent program [35] that is bounded by the same constraints as the original program. The goal of such a transformation is to enhance and improve the original program, whether the improvement be scalability, efficiency, parallelization or some other measure of improvement. This goal exists as programming, in any language, can be an arduous task, with many aspects that have to be taken in to consideration by the developer.

There are two main approaches to the transformation of functional programs. The first approach is fold/unfold based systems, based on the works of Burstall and Darlington [7]. An unfolding is where a function call within an expression is replaced with a corresponding instance of the function body, and a folding is replacing an instance of a function body with a corresponding call to the function. In fold/unfold based systems, a function call is typically unfolded, and then the resulting expression is optimized according to a set of transformation rules.

The second approach is calculational methods based systems, based on the Bird-Meertens Formalisms (BMF), which guarantee the correctness of the resulting program [3, 2, 4, 31, 14, 1, 37]. In such systems, a series of calculational laws, describing a program's properties, are applied to a program in order to transform it into another. These calculational laws are usually combined into a calculational algorithm describing the transformation process [40].

While we focus on program transformation applied to functional languages in this paper, it is worth noting that program transformation is applicable to other types of language, such as logic languages [30], and sequential languages [29]. There are a number of reasons why we focus on functional languages, but the most important reasons are that functional languages are easier to analyze, reason about, and to manipulate using program transformation techniques. The lack of side-effects in pure functional languages is a major benefit, as these do not have to be taken into consideration during the transformation process.

This lack of side-effects is also beneficial in the parallelization of programs, as functions are stateless as a result, and therefore one process executing a function on a set of data can have no impact on another process executing the same function on another set of data, thereby giving functional programs an implicit parallelism. Another benefit is that as execution order is constrained solely by data dependencies, and not statement order, sub-expressions may also be executable in parallel, as long as there are no data dependencies between them. In the case of where there are data dependencies between processes, for example, process a depends on some data in process b, then a can be suspended until b has completed, and then a can be resumed. An obvious problem with this is that it can lead to a lot of waiting, and context switching, which can have a negative impact on execution performance. Another problem with using functional languages and lazy evaluation is that the only expressions that should be evaluated are those that are needed for the overall computation. To ensure that only the expressions that are needed are evaluated, a strictness analysis may have to be performed.

Automatic parallelization can be of huge benefit to developers. If a developer can write a sequential solution to a given problem, and have that automatically transformed into a parallelized version that is optimized based upon the underlying hardware that the resulting transformed program will be deployed on, it eliminates all need for the developer to worry about the parallelization issues. All the developer has to concentrate on is writing a sequential version that solves the problem. This can then be optimized using one of the transformation techniques listed above, and then parallelized,

so the developer can automatically have an optimized, parallelized version of the sequential version previously implemented.

2 Language

For the purposes of this paper, a representation of a language and its functions is needed to examine the discussed techniques. These conventions are based on the Bird-Meertens Formalisms [3, 2, 31, 14, 1, 37], which allows for concise descriptions of program transformations and is a good architecture-dependent parallel model [38]. Function application, denoted by juxtaposition can be written with or without brackets, therefore $f\ x \equiv f\ (x)$. Functions are curried, the binding of function application is stronger than any other, and application is left associative, $f\ x\ y \equiv (f\ x)\ y$. Function composition is denoted $\circ$, and as expected, $(f \circ g)\ x \equiv f\ (g\ x)$.

Lists are also an important part of this language, and are detailed below. As expected, the concatenation of two lists joins two lists and is associative. While a list is essentially a list, when comparing sequential and parallel versions of programs, we must differentiate them in some way and this is dealt with in the definition of lists below:

Sequential (*cons*) List There are two possible types of list in the sequential sense, an empty list, $[]$, and a list concatenation, containing an element x, and another list xs, denoted $(x : xs)$.

Parallel (*join*) List In the parallel sense, there are three possible types of lists, an empty list, $[]$, a singleton containing the element x, denoted $[x]$, and a list concatenation, in which a list, xs, is joined with another list ys, $xs +\!\!+ ys$.

The Bird-Meertens Formalisms also contain some very useful, commonly used higher order functions, such as *map*, *reduce* and *scan*, among others. These functions are shown below, and are very important when used to aid parallelization, for reasons which will be explained in later sections. By expanding the *reduce* function, using the addition and concatenation operators, $(+)$ and $(+\!\!+)$ respectively, we can generate functions to sum or flatten a list [10].

$$
\begin{array}{ll}
map\ f\ [x_1, x_2, \ldots, x_n] & = [f\ x_1, f\ x_2, \ldots, f\ x_n] \\
reduce\ \oplus\ [x_1, x_2, \ldots, x_n] & = x_1\ \oplus\ x_2\ \oplus \ldots\ \oplus\ x_n \\
scan\ \oplus\ [x_1, x_2, \ldots, x_n] & = [x_1, x_1\ \oplus\ x_2, \ldots, x_1\ \oplus\ x_2\ \oplus\ \ldots \oplus\ x_n] \\
sum & = reduce\ (+) \\
flatten & = reduce\ (+\!\!+)
\end{array}
$$

Fig. 1. Example Functional Skeletons

3 Skeletons

Skeletons, due to Cole [9], are well recognized common patterns of parallel programming that can be efficiently mapped to parallel architectures without involving the developer of a program that uses them. These primitive functions can be used to assist the development of parallel versions of sequential programs, and may be viewed as the building blocks of a parallel program. These

building blocks should be viewed as higher-order functions, each of which have a known efficient parallel implementation. *map*, *reduce* and *scan*, shown above are very important skeletal functions, and some further examples are shown in Figure 2 below.

$$\begin{aligned} zip\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] &= [(x_1, y_1), \ldots, (x_n, y_n)] \\ zipWith\ f\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] &= [(f\ x_1\ y_1), \ldots, (f\ x_n\ y_n)] \end{aligned}$$

Fig. 2. Example Functional Skeletons

Cole [9] defines a framework in which, from a set of known skeletons, the developer of a program selects one, which is to be used as the 'building block' of their program, and completes the rest of their program inside this block. Inside this block, all features of the language remain available to the developer, and an efficient parallel program can be derived from the sequential code within the selected skeleton.

Using skeletons is beneficial because the skeletons show how the program can be decomposed into parallel parts, and describe the strategy for parallelizing the program. This removes the need for the developer to decompose a sequential program into its parallel equivalent, however, one problem with skeletons is that as parallel architectures differ, so do their underlying implementations. This means that as architectures change, the skeletons themselves may need new definitions. However, as with the underlying mechanics of, for example, a compiler being different on different architectures, this should not be of concern to the developer.

Despite the advantages of the skeleton approach, developing efficient parallel programs still remains a big challenge for two reasons. Firstly, as is pointed out in [39], it is hard to choose appropriate parallel skeletons for a particular problem, especially when the given problem is complicated. This is due to the gap between the simplicity of parallel skeletons and the complexity of the algorithms to be parallelized. Secondly, as is pointed out in [32], although a single skeleton can be efficiently implemented, a combination of skeletons may not execute efficiently. This is because skeletal parallel programs tend to introduce many intermediate data structures for communication among skeletons. To achieve good results, we have to eliminate unnecessary creations and traversals of such data structures, and unnecessary inter-process communications (exchanges of data).

4 Calculational Approaches to Program Parallelization

4.1 List Homomorphisms & Near-Homomorphisms

Each of the aforementioned skeletons is important in the works on deriving parallelism using calculational methods, as they are what are known as *list homomorphisms*. The use of such list homomorphisms, due to Bird [3] as an approach to parallelization was first suggested by Skillicorn [37]. The use of these list homomorphisms and the skeletal functions they are composed of is due to the skeletons being data parallel in nature, and removing the issues surrounding parallelization from the developer.

Definition (List Homomorphism) A function f is defined as a list homomorphism if it matches the following definition, where $\oplus$ is an associative binary operator.

$$f\ [x] \qquad = g\ x$$
$$f\ (xs +\!\!+ ys) = f\ xs \oplus f\ ys$$

$(\!|g, \oplus|\!)$ denotes the unique function f[25]. □

It is obvious from this that the computations of $f\ xs$ and $f\ ys$ are independent and can be done in parallel [17]. It is important to note, that according to the homomorphism lemma, defined below, any list homomorphism can be written as a composition of a *map* and a *reduce*, and that both *map* and *reduce* are list homomorphisms themselves [3]. This is very important in terms of parallelism, as it means that every list homomorphism has a corresponding composition (simple diffusion [26]) of *reduce* and *map* [3], both of which, along with *scan* have known highly efficient parallel implementations [38, 6].

Lemma (List Homomorphism) Any function f is a list homomorphism, with respect to $(+\!\!+)$, iff, for some function g, and a binary associative operator $\oplus$, the following holds:

$$f = (\!|g, \oplus|\!) = (reduce\ \oplus) \circ (map\ g)$$

□

Based upon these homomorphisms are the promotion lemmas, defined below, which state that some homomorphisms, *map* and *reduce* specifically can be defined in terms of themselves [3] where the reduction of the homomorphism is on $(+\!\!+)$. Essentially this means that when mapping or reducing over a list, the list can be segmented, and that operation applied to each of the segments, and the results of each combined. Again, this is of particular merit as each application of the homomorphism to each segment can be executed in parallel. This combined with the fact that homomorphisms are parallel in nature themselves leads to opportunities for greater parallelism.

Lemma (Homomorphism Promotion) Given a function f, and an associative binary operator $\oplus$, the promotion lemmas are defined below:

$$(map\ f) \circ (reduce\ (+\!\!+)) \quad = (reduce\ (+\!\!+)) \circ (map\ (map\ f))$$
$$(reduce\ \oplus) \circ (reduce\ (+\!\!+)) = (reduce\ \oplus) \circ (map\ (reduce\ \oplus))$$

□

Some of the reasons for selection of BMF as a parallel model in [37] are that as the transformations are calculational in nature, the program's correctness is preserved, and there is an obtainable optimal solution. The resulting programs can also be easily understood as they are sequential in nature but parallel in data, and decomposition and parallelization is taken care of in the skeletal functions.

However, the methodology presented by Skillicorn is quite restrictive, as while it does provide numerous equations, these are all defined in terms of *map* and *reduce*. There are some other operations provided, such as *scan*, shown above, but again this is derived from *map* and *reduce*. It would be quite a challenging task to define many programs just in terms of *map* and *reduce*, when there are other more appropriate solutions available.

Works up to this point [38, 37] were restricted to a class of programs that contained list homomorphisms, but this restriction excludes a whole other class of programs that contain functions that are almost, but not quite homomorphisms. Cole [10] expands upon this restriction, and introduces the idea of a near-homomorphism.

A 'near' homomorphism, defined below, is a function that when combined, or tupled with an auxiliary function, or projection, can be converted into a homomorphism by means of composition. This is an important method, as it opens up the parallelism available to list homomorphisms to this class of functions that are not quite homomorphisms. It is also important to note that any function can be made homomorphic by tupling it with the identity function [18]. When the projection is combined with the homomorphism derived from the original function, it should be equivalent to the original function.

Definition (Near-homomorphism) Given a function that is not quite a homomorphism, g, a homomorphism f, and a projection π, a near homomorphism is defined as:

$$g = \pi \ \circ \ f$$

□

4.2 Distributable Homomorphisms

Gorlatch [17] attempts to provide another method for deriving efficient parallel programs, from recursive function definitions, using BMF, by introducing a new class of homomorphism, *distributable homomorphisms* (DHs) and an efficient and correct parallel schema for implementing these functions. Again, Gorlatch selected BMF as a derivational calculus due to its guarantee of correctness upon transformation, and DHs are an attempt to solve two main problems with using list homomorphisms to guide the parallelization process: that of finding a suitable $\oplus$, the combination operator for the homomorphism, and efficiently implementing the reduction part in parallel [17].

Within this definition of the combination operator, the class of problems that it is applicable to is restricted in nature, as can be seen within the definition below; however there is implicit parallelism within the definition. As $zipWith$ is applied over two separate associative operators to two lists, they can be completed in parallel, and then concatenated together.

Definition (Distributable Homomorphism) A function f is a distributable homomorphism, for given associative operators $\oplus$ and $\otimes$, if its definition matches that of *dist* below:

$$\begin{aligned} dist \ \oplus \ \otimes \ [x] \ \ \ \ \ \ \ \ \ \ &= [x] \\ dist \ \oplus \ \otimes \ (xs +\!\!+ ys) &= combine \ \oplus \ \otimes \ (dist \ \oplus \ \otimes \ xs) \ (dist \ \oplus \ \otimes \ ys) \end{aligned}$$

$$combine \ \oplus \ \otimes \ xs \ ys = (zipWith \ \oplus \ xs \ ys) +\!\!+ (zipWith \ \otimes \ xs \ ys)$$

□

As an example, if we consider a distributed reduction function, over an associative operator, $\odot$, similar in nature to $MPI\_ReduceAll$ of the C MPI standard, defined like so:

$$dReduce \ \odot \ x = [reduce \ \odot \ x, reduce \ \odot \ x, ..., reduce \ \odot \ x]$$

this is obviously a homomorphism, and is also a distributable homomorphism where the two associative operators are both $\odot$.

The author presents a derivation of *scan* into the distributable homomorphism format shown above, via tupling, and projections. However, while this is interesting, there is no attempt made to show how this process could be automated, or how to extract the associative operators needed.

While this is obviously a powerful class of homomorphism, and very parallel in nature, automation of this process would require some additional information. As distributive homomorphisms are dependent on associative operators, a process for extracting these operators is needed, and also a process for deriving the distributable homomorphisms themselves is needed. However, also shown in [17] is a guide to efficient parallel implementations of distributable homomorphisms on a hypercube architecture, which does indeed show that optimal implementations can be obtained on the architecture level.

Gorlatch presents another derivational method for extracting parallelism via homomorphisms in [18], based on the work of Gibbons on the third homomorphism theorem [15], a folk theorem [21], that states that any function on a list that can be calculated both in a leftward and rightward direction is a list homomorphism. This third homomorphism theorem is motivated by the second homomorphism theorem [3], which states that any list homomorphism can itself be calculated in a leftward or rightward fashion.

To understand the third homomorphism theorem, we first need a definition of both a leftward and rightward function. These definitions are given below.

Definition (Leftwards function) For a binary operator $\oplus$ - not necessarily an associative function - a function, f is $\oplus$-leftwards, iff, for all elements x and lists xs.

$$f\ ([x] +\!\!+ xs) = x\ \oplus\ f\ xs$$

- Where $f\ [] = e$, f's $\oplus$-leftwards function is $foldr\ \oplus\ e$.
- Also, $foldr\ \oplus\ e\ (xs +\!\!+ ys)$ can be written $foldr\ \oplus\ (foldr\ \oplus\ e\ ys)\ xs)$

□

Definition (Rightwards function) For a binary operator $\otimes$ - not necessarily an associative function - a function, f is $\otimes$-rightwards, iff, for all lists xs and elements x.

$$f\ (xs +\!\!+ [x]) = f\ xs \otimes\ x$$

- Where $f\ [] = e$, f's $\otimes$-rightwards function is $foldl\ \otimes\ e$.
- Also, $foldl\ \otimes\ e\ (xs +\!\!+ ys)$ can be written $foldl\ \otimes\ (foldl\ \otimes\ e\ xs)\ ys)$

□

Given these definitions of leftwards and rightwards functions, the third homomorphism theorem is as follows.

Definition (Third Homomorphism Theorem) If a function is both leftwards and rightwards in nature, then it is a homomorphism. □

Gibbons [15] provides an example derivation - deriving *mergesort* from *insertionsort* - to show the utility of this fact. However, and the author notes this, the derived homomorphism is not efficient and more work needs to be done to derive efficiency.

As stated above, Gorlatch builds upon this theorem in [18], in which the author generates list homomorphisms via generalization of both leftward and rightward functions, that for some non-homomorphic functions can provide embedding into homomorphisms. The notion of both leftward and rightward functions is expanded upon to allow for left and right-homomorphic functions, definitions of which are obvious.

Definition (Left and Right-Homomorphism Theorem) Where a function, f, with a not necessarily associative combination operation $\ominus$, is a homomorphism, it is both a left and right-homomorphism with $\ominus$. If a function, g, is a left or right-homomorphism, and its combination operation, $\ominus$, is associative, then it is a homomorphism with $\ominus$. □

This theorem provides a means to deriving a homomorphic version of the function; by deriving a cons-list version of the function that is a left-homomorphism, and then proving that its combination operation is associative, it becomes a homomorphism. However, this derivation method is not always successful and to solve this problem, the author details a synthesis method: by generalizing both a function's leftward and rightward terms, into another term, t' that defines a combination operation $\ominus$. If this generalized term can be obtained, and $\ominus$ is associative, then the original function is a homomorphism, and it's combination operation is $\ominus$.

This idea is then applied to near-homomorphisms, and the main difficulty associated with them - determining the auxiliary functions that they must be combined with and their combination operation. To allow for this, the homomorphism definition and the generalization method are both extended. The homomorphism definition is simply extended to allow for tuples, and support for the union of tuples is added to the generalization method. The generalization method is then applied in a pairwise fashion to the tuple constructed when adding in the necessary auxiliary functions.

Using the distributable homomorphism again, this work is further extended to aid in the efficient parallel schemas of the resulting homomorphisms on a parallel architecture, a known problem with homomorphisms. However, there is another problem with this as it requires that the definition of a function be given in both a leftward and rightward form, which is restrictive in nature, and further work [25] shows that derivation of a parallel implementation only requires a leftward definition.

4.3 List Mutumorphisms

Hu. et. al. [25] further previous work, by introducing a derivation system that is applicable to a broad class of primitive recursive functions. In this paper the authors attempt to free developers from the constraint of having to write programs in terms of a small set of skeletons. By targeting these general recursive forms, and by supplying additional BMF parallelization rules and theorems, that are more powerful than previous works [19, 37, 17, 18] and a new inductive synthesis lemma, they create a new systematic parallelization algorithm to derive these parallel programs. As with previous works, their algorithm is based in program calculation and is guaranteed to be correct and terminate, and can in fact be generalized to work for not just lists, but other more complex datatypes.

Tuples play an important part in this work, and as such need to be more formally defined at this point than previously, specifically, a function is introduced, π_i, a projection that selects the i^{th} element of a tuple. A binary operator Δ is also introduced that applies a function to a tuple, such that $(f \ ^{\Delta} \ g) \ t = (f \ t, g \ t)$, and $f_1^{\Delta} \ \ldots \ ^{\Delta} \ f_n$ can be denoted by $\Delta_1^n \ f_i$.

To allow this more powerful parallelization method, the authors look beyond homomorphisms, near-homomorphisms and distributed homomorphisms to *list mutumorphisms*, as a more powerful parallel model. List mutumorphisms [14] are a better parallel model, as they are more descriptive, and as a result more powerful [14], and can be converted to list homomorphisms automatically, using the tupling calculation defined previously by Hu. et. al. [24].

Definition (List Mutumorphism) The functions $f_1, ...f_n$ are list mutumorphisms, if they are defined as shown below.

$$f_j\ [x] \qquad\quad = g_j\ x$$
$$f_j\ (xs +\!\!+ ys) = ((\Delta_1^n\ f_i)\ xs) \oplus ((\Delta_1^n\ f_i)\ ys)$$

In particular f_i is a list mutumorphism if there exist functions $f_1, ..., f_{i-1}, f_{i+1}, ...f_n$, that when combined with f_i satisfy the above form. □

Definition (Tupling Property) If $h_1, ..., h_n$ are mutumorphisms as defined above, then:

$$\Delta_1^n\ f_i = (\!|\Delta_1^n g_i, \Delta_1^n \oplus_1|\!)$$

And, following from this, any mutumorphism f can be transformed into a composition of a projection and a homomorphism:

$$f_j = \pi_j \circ (\!|\Delta_1^n g_i, \Delta_1^n \oplus_1|\!)$$

□

After their parallelization process is complete, the definition of f should be in the form of a mutumorphism:

$$f\ (xs +\!\!+ ys) = \ \ldots\ f\ xs\ \ \ldots\ f\ ys$$

At the core of their parallelization technique is their synthesis lemma [25], in which two well known useful [41] synthesis techniques *generalization* and *induction* are embedded.

Definition (Calculational Law) Given an equation of the form, where $\oplus$ is associative:

$$f\ (x : xs) = f\ [x] \oplus f\ xs$$

The following parallel definition can be obtained:

$$f\ (xs +\!\!+ ys) = f\ xs \oplus f\ ys$$

□

The authors feel that the basis of previous works - the calculational law above - is not adequate and propose four extensions to it and Bird's laws, firstly removing restrictions on the parameters of the given function, then allowing an accumulating parameter, thirdly by allowing conditional structures and lastly allowing non-linear recursion. Following these extensions, the authors then present their parallelization theorem.

Definition (Parallelization Theorem) Given a function, f, of the following form:

$$f\ (x : xs)\ \{c\} = e_1 \oplus ... \oplus e_n$$

where

– $\{c\}$ denotes optionality.

where g_i is a function and q_i is a mutumorphism and where each e_i is one of the following:

– a non-recursive expression: $g_i\ a\ (q_i\ x)\ \{c\}$
– a recursive call: $f\ x\ \{g \odot c\}$ where g is a function and $\odot$ is associative

– a conditional expression wrapping recursive calls: **if** $g_{i_1}\ a\ (q_{i_1}\ x)\ \{c\}$ **then** $e'_1 \oplus ... \oplus e'_n$ **else** $g_{i_2}\ a\ (q_{i_2}\ x)\ \{c\}$, where at least one e'_i is a recursive call and the others are non-recursive terms

Functions matching this form can be parallelized by calculation, as long as all function calls to f are the same, and where there are more than two, $\oplus$ is commutative, and $\otimes$ exists and is distributive over $\oplus$. □

The authors then present their parallelization algorithm, an important step of which is identifying associative and distributive operators, as they are key to the above extensions and the parallelization theorem. There are many ways in which these can be identified [41, 13, 22], but the authors are able to derive this information based on the type of the function to be parallelized, based on the work of Sheard and Fegaras [36], that in short, if given a function f with type $\mathcal{R}$, $\oplus$ is $\mathcal{R}$'s zero replacement function, and based on this, a distributive $\otimes$ can easily be defined.

Within the algorithm, there are three steps: making the associative operator explicit, using the theory above, then normalizing the body by abstraction and fusion, and finally applying the parallelization laws and optimizing the tupling calculation. The algorithm results in a parallel program in a mutumorphic form and may need some further optimization using an automatic tupling calculation shown earlier in the work.

This is obviously quite a powerful transformational method, as it applies to a very general form of recursive programs, and is not restricted to any specific type of recursive form. It can handle extra information over previous works, such as non-linear recursive functions, accumulating parameters and conditional statements, and uses type information to derive associative operators.

4.4 Diffusion

In [26], Hu et. al., attempt to build upon previous work [25, 3, 11] in a technique they present called *diffusion*, generalizing the homomorphism lemma to allow for accumulating parameters. As a result, diffusion is applicable to a broad class of recursive functions, and if combined with the normalization technique mentioned previously [25] is applicable to an even broader class of programs. The authors also highlight how diffusion can be generalized to work with trees and general data structures, known to be a difficulty in data parallel programming [28, 34]. Once a function definition is in the form required by diffusion, it can then be automatically transformed into an efficient parallel version.

At the core of diffusion is the diffusion theorem, which applies to three classes of program, the first of which is those that can simply be diffused to *reduce*, the second of which can diffused to a composition of *reduce* and *map*, and the last of which can be diffused to a combination of *reduce*, *map* and *scan*. This latter one is the most interesting of the three, as the other two have been dealt with before. This type of diffusion attempts to solve the limitation where an application of a homomorphism to a list results in a function instead of a value, and focuses on functions with an accumulating parameter and that match the following form:

$$\begin{aligned} f\ []\ c &= g_1\ c \\ f\ (x : xs)\ c &= k\ (x, c)\ \oplus\ (f\ xs\ (c\ \otimes\ g_2\ x)) \end{aligned}$$

Hu. et. al. get around the problem of the accumulating parameter by precomputing all values of it, and then making them appear as a constant. This leads on to the diffusion theorem.

Definition (Diffusion Theorem) Using the above recursive form, and the fact that $\oplus$ and $\otimes$ are associative and have units, a recursive function, f, is diffusible into the following:

$$
\begin{array}{lll}
f\ x\ c = \textbf{let}\ cs' +\!\!+ [c'] & = map\ (c\otimes)\ (scan\ \otimes\ (map\ g_2\ x)) \\
\quad\quad ac & = zip\ x\ cs' \\
\quad \textbf{in}\ \ (reduce\ \oplus\ (map\ k\ ac)) \oplus (g_1\ c') &
\end{array}
$$

□

Using the diffusion theorem, and recursive function definitions above, the authors present their algorithm for diffusion. While their recursive definition is useful, a lot of functions that could be diffused do not match this definition initially and must be transformed into diffusible form. There are four steps to this algorithm: linearizing recursive calls, identifying associative operators, applying the diffusion theorem and optimizing operators. This is a very powerful approach to deriving parallel code, as it can handle accumulating parameters, and is applicable to all primitive recursive functions, and it identifies associative operators prior to deriving a parallel version.

As an example, if we consider a naive solution to the bracket matching problem [11] that matches brackets using a stack, as defined below:

$$
\begin{array}{lll}
bm\ []\ s & = isEmpty\ s \\
bm\ (x:xs)\ s & = \textbf{if}\ isOpen\ a\ \textbf{then}\ bm\ xs\ (push\ x\ s) \\
& \quad \textbf{else if}\ isClose\ x\ \textbf{then}\ noEmpty\ s\ \wedge match\ x\ (top\ s)\ \wedge\ bm\ xs\ (pop\ s) \\
& \quad \textbf{else}\ bm\ xs\ s
\end{array}
$$

As the definition of bm contains more than one recursive call, it is non-linear and needs to be linearized, the result of which is shown below.

$$
\begin{array}{lll}
bm\ []\ s & = isEmpty\ s \\
bm\ (x:xs)\ s & = g_1\ x\ s\ \wedge\ bm\ xs\ (g_2\ x\ s)
\end{array}
$$

$$
\begin{array}{lll}
g_1\ x\ s & = \textbf{if}\ isOpen\ x\ \textbf{then}\ True \\
& \quad \textbf{else if}\ isClose\ x\ \textbf{then}\ noEmpty\ s \wedge\ match\ x\ (top\ s) \\
& \quad \textbf{else}\ True
\end{array}
$$

$$
\begin{array}{lll}
g_2\ x\ s & = \textbf{if}\ isOpen\ x\ \textbf{then}\ push\ x\ s \\
& \quad \textbf{else if}\ isClose\ a\ \textbf{then}\ pop\ s \\
& \quad \textbf{else}\ s
\end{array}
$$

Once the function has been linearized, then the next step is to identify the associative operators. As $\oplus$ is associative, and should be associative over the domain of the function, it is obvious that a suitable $\oplus$ for bm is $\wedge$. Finding a suitable $\otimes$ is more complicated, but following [26], Hu. et. al. derive the following operator for combining two stacks:

$$
\begin{array}{lll}
s \otimes Empty & = s \\
s \otimes (Push\ x\ s') & = Push\ x\ (s \otimes s') \\
s \otimes (Pops') & = Pop\ (s\ \otimes\ s')
\end{array}
$$

And using this $\otimes$, g_2 becomes:

$$
\begin{array}{lll}
g_2\ x\ s & = s\ \otimes\ g_2'\ x \\
g_2'\ x & = \textbf{if}\ isOpen\ x\ \textbf{then}\ Push\ x\ Empty \\
& \quad \textbf{else if}\ isClose\ x\ \textbf{then}\ Pop\ Empty \\
& \quad \textbf{else}\ Empty
\end{array}
$$

After finding appropriate associative operators for the function, the next step is to apply the diffusion theorem resulting in the following definition of bm:

$$\begin{aligned} bm\ x\ c = \textbf{let}\ & cs' +\!\!+ c' = map\ (c \otimes)\ (scan\ \otimes\ (map\ g_2\ x)) \\ & ac = zip\ x\ cs' \\ \textbf{in}\ & (reduce\ (\wedge)\ (map\ g_1\ ac)\ \wedge\ isEmpty\ c') \end{aligned}$$

After applying diffusion, the final step is to optimize the operators used within the function definitions. For brevity's sake, we do not detail this process here.

4.5 Accumulate Skeleton

Iwasaki and Hu build upon their above previous work in [27], developing a new skeleton - the *accumulate* skeleton - that can be used to intuitively derive solutions to problems, and that aims to solve one of the main problems when working with skeletons; that of selecting the correct skeletons and the combination of them. As *accumulate* is derived from diffusion, it is also applicable to the broad class of programs that can be diffused, and is more descriptive than previous skeletons such as *scan*. Some benefits of *accumulate* are that it uses *fusion* [16, 5] to eliminate intermediate data, and can also eliminate multiple accesses to data, and decreases communication between processors.

Definition (Accumulate Skeleton) Where g, p, q are functions and $\oplus$ and $\otimes$ are associative operators, the *accumulate* skeleton, denoted $[\![g, (p, \oplus), (q, \otimes)]\!]$, is defined as:

$$\begin{aligned} accumulate\ []\ c &= g\ c \\ accumulate\ (x : xs)\ c &= p\ (x, c) \oplus\ accumulate\ xs\ (c \otimes q\ x) \end{aligned}$$

And due to diffusion, can be written in terms of *reduce*, *map*, *scan* and *zip* as follows:

$$\begin{aligned} [\![g, (p, \oplus), (q, \otimes)]\!]\ xs\ c\ &= reduce\ (\oplus)\ (map\ p\ as)\ \oplus\ g\ b \\ \textbf{where}\ bs +\!\!+ [b] &= map\ (c \otimes)\ (scan\ \otimes (map\ q\ xs)) \\ as\ &= zip\ xs\ bs \end{aligned}$$

□

As an example, we use a function that eliminates the smaller elements of a list, *smaller*, defined below:

$$\begin{aligned} smaller\ []\ c &= [] \\ smaller\ (x : xs)\ c &= (\textbf{if}\ x < c\ \textbf{then}\ []\ \textbf{else}\ [x]) +\!\!+ smaller\ xs\ (\textbf{if}\ x < c\ \textbf{then}\ c\ \textbf{else}\ x) \end{aligned}$$

As stated above, *accumulate* is derived using diffusion. The result of applying diffusion to *smaller* is shown below:

$$\begin{aligned} & smaller\ xs\ c\ = reduce\ (+\!\!+)\ (map\ p\ as)\ +\!\!+\ g\ b \\ & \textbf{where} \\ & \quad bs\ +\!\!+\ [b] = map\ (c \otimes)\ (scan\ (\otimes)\ (map\ q\ xs)) \\ & \quad as\ = zip\ xs\ bs \\ & \quad c\ \otimes\ x\ = \textbf{if}\ x < c\ \textbf{then}\ c\ \textbf{else}\ x \\ & \quad p\ (x, c)\ = \textbf{if}\ x < c\ \textbf{then}\ []\ \textbf{else}\ [x] \\ & \quad g\ c\ = [] \\ & \quad q\ x\ = x \end{aligned}$$

This can then be rewritten using the *accumulate* skeleton as:

$$
\begin{array}{ll}
smaller\ xs\ c & = [\![g, (p, +\!\!+), (q, \otimes)]\!]\ xs\ c \\
\textbf{where} & \\
\quad c\ otimes\ a & = \textbf{if}\ a < c\ \textbf{then}\ c\textbf{else}\ a \\
\quad p\ (x, c) & = \textbf{if}\ x < c\ \textbf{then}\ []\ \textbf{else}\ [x] \\
\quad g\ c & = [] \\
\quad q\ x & = x
\end{array}
$$

The benefits of using the *accumulate* skeleton should be obvious, as it allows quite a general class of program to be parallelized, and is quite intuitive. Obviously as this skeleton is itself composed of more skeletons, each of which are highly parallel, there is quite a broad scope for parallelization involved. Defining programs in terms of *accumulate* should be a reasonable process, as the developer need only identify the parameters needed by $[\![g, (p, \oplus), (q, \otimes)]\!]$. However a difficulty associated with this would be finding suitable associative operators for $\oplus$ and $\otimes$, however the context preservation transformation in [8] can provide a solution for this.

4.6 Deriving Homomorphisms Using a Weak Right Inverse

Morita et. al. [33] also present another new parallel derivation approach in their work, again based on the third homomorphism theorem that derives homomorphisms from a pair of sequential programs, as the third homomorphism theorem states that if two sequential programs - in a specific form - exist to solve a problem, then a list homomorphism exists too. To derive optimal homomorphisms, the authors use an automatic derivation of the *weak right inverse* of functions, and show that this always exists, and is applicable to a broad class of sequential functions. One of the key principles of the derivation is that users will define their functions in terms of sequential functions, and the authors guarantee that under reasonable conditions an efficient parallel version can be derived.

Definition (Weak Right Inverse) A function f° is a weak right inverse of function f if it satisfies the following:

$$\forall b \in range(f), f^\circ\ b = a \Rightarrow f\ a = b$$

where $range(f)$ denotes the range of function f. □

f° is a weak right inverse as the domain of f° can be larger than the range of f, where as if it were just a right inverse, it's domain would be within the range of f. According to [33], two important points about weak right inverses are that at least one exists for every function, and that the following property holds for each weak right inverse: $f \circ f^\circ \circ f = f$. These two points are important because, when combined with the fact that if a function g exists where $f \circ g \circ f = f$, then an associative operator can be derived. The importance of this should be obvious, and is the basis of the following theorem.

Definition (Parallelization with Weak Right Inverse Theorem) If a function h is both leftwards and rightwards, then the following holds:

$$
\begin{array}{l}
h = (\!|f, \odot|\!) \\
\textbf{where} \\
\quad f\ a\ \ = h\ [a] \\
\quad a \odot b = h\ (h^\circ\ a +\!\!+ h^\circ\ b)
\end{array}
$$

□

As an example, consider the function *sum*, that calculates the sum of a list. It is obvious that this function is both leftwards and rightwards, and its weak right inverse, $sum^\circ = [x]$ as shown below:

$$\begin{aligned} sum\ [x] &= x \\ sum\ ([x] +\!\!+ xs) &= x + sum\ xs \\ sum\ (xs +\!\!+ [x]) &= sum\ xs + x \end{aligned}$$

Using the above theorem, the following homomorphism for *sum* can be derived:

$$\begin{aligned} &sum = (\!|f, \odot|\!) \\ &\textbf{where} \\ &\quad f\ a = a \\ &\quad a \odot b = a + b \end{aligned}$$

The inherent parallelism in this method should be obvious, and it shows that all that is needed to derive a list homomorphism is to derive a functions weak right inverse. Morita et. al.'s parallelization algorithm is based upon this, and allows for automatic derivation of parallel skeletal programs from sequential ones. Their parallelization algorithm has four steps, the first of which is to get the constraint equations for an input list, of fixed length, by unfolding sequential equations. The next step is to solve these constraints and derive a possibly inefficient weak right inverse. This is then optimized through the removal of unnecessary conditional branches. Finally, verification is done on the domain of the weak right inverse to ensure it satisfies preconditions.

This parallelization system is a powerful and promising one, as it can derive homomorphisms and therefore parallelization for a very broad set of functions. However, it is constrained by the assumption that the output of a weak right inverse will be of fixed length. This assumption excludes a large number of functions from being parallelized, and while this issue can be solved by generalization, it hasn't been resolved yet.

5 Conclusions

In conclusion, we have covered many worthy parallelization techniques here. We have detailed the Bird-Meertens Formalisms [3, 2], and skeletons [9] and how they provide the basis for these techniques. We have covered list homomorphisms [37], near homomorphisms [10], distributable homomorphisms [17, 18], list mutumorphisms [25], diffusion [26], the *accumulate* skeleton [27] and a means of deriving homomorphisms using the weak right inverse of a function [33]. Indeed, homomorphisms and their derivatives are really the basis of these parallelization techniques, and each successive work generally encompasses a broader class of program that can be parallelized.

Also key to these parallelization techniques are the underlying skeletal patterns that are used; *map*, *reduce* and *scan*. These are quite powerful in themselves and they have known efficient parallel implementations, and when combined with parallelization techniques such as those presented here they provide the basis for the efficient parallelization of programs. Also worth noting is that due to the calculational methods used in parallelization, this allows a program developer to develop sequentially, and for a parallel version to be derived from this sequential version.

While we find these transformation techniques to be quite powerful, they do involve quite complex manipulations to convert input programs to a form that is parallelizable. A technique that was

applicable to a more commonly known language, even if restricted, would be more user friendly, and may allow for some more general transformations, such as the fold/unfold methodology of Burstall and Darlington [7]. This technique has proven quite useful in optimizing sequential higher-order functional programs. While there is obviously quite a lot of work done on deriving parallelism using calculational methods, we find there to be room for research on deriving parallelism using a fold/unfold methodology that can be fully automated. For instance, distillation [20] has been shown to not only achieve a super-linear speed increase in the efficiency of sequential programs, but has also been shown to convert programs into a form which is more amenable to parallelization; in fact the output is of the form described by the accumulate skeleton [27]. We intend to pursue this approach to the parallelization of general functional programs using the fold/unfold methodology.

6 Acknowldgements

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre

References

1. Backhouse, R.: An exploration of the bird-meertens formalism. Tech. rep., In STOP Summer School on Constructive Algorithmics, Abeland (1989)
2. Bird, R.: Constructive functional programming. STOP Summer School on Constructive Algorithmics (1989)
3. Bird, R.S.: An introduction to the theory of lists. In: Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design. pp. 5–42. Springer-Verlag New York, Inc., New York, NY, USA (1987)
4. Bird, R.S.: Algebraic identities for program calculation. The Computer Journal 32(2), 122–126 (1989)
5. Bird, R.: Introduction to Functional Programming using Haskell. Prentice Hall PTR, 2 edn. (May 1998)
6. Blelloch, G.E.: Scans as primitive operations. IEEE Transactions on Computers 38(11), 1526–1538 (1989)
7. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the Association for Computing Machinery 24(1), 44–67 (January 1977)
8. Chin, W.N., Khoo, S.C., Hu, Z., Takeichi, M.: Deriving parallel codes via invariants. In: Palsberg, J. (ed.) Static Analysis, Lecture Notes in Computer Science, vol. 1824, pp. 59–65. Springer Berlin / Heidelberg (2000)
9. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. Ph.D. thesis, Department of Computing Science, University of Glasgow (1989)
10. Cole, M.: Parallel programming, list homomorphisms and the maximum segment sum problem. Tech. rep., Proceedings of Parco 93. Elsevier Series in Advances in Parallel Computing (1993)
11. Cole, M.: Parallel programming with list homomorphisms. Parallel Processing Letters 5, 191–203 (1995)
12. Culler, D.E., Sing, J.P., Gupta, A.: Parallel Computer Architecture - A Hardware/Software Approach. Morgan Kaufmann Publishers (1999)
13. Fisher, A.L., Ghuloum, A.M.: Parallelizing complex scans and reductions. SIGPLAN Not. 29, 135–146 (June 1994)
14. Fokkinga, M.: In: Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics
15. Gibbons, J., Gibbons, J.: The third homomorphism theorem (1995)
16. Gill, A., Launchbury, J., Jones, S.P.: A shortcut to deforestation. FPCA: Proceedings of the conference on Functional programming languages and computer architecture pp. 223–232 (1993)

17. Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In: In Annual European Conference on Parallel Processing, LNCS 1124. pp. 401–408. Springer-Verlag (1996)
18. Gorlatch, S.: Systematic extraction and implementation of divide-and-conquer parallelism. In: Programming languages: Implementation, Logics and Programs, Lecture Notes in Computer Science 1140. pp. 274–288. Springer-Verlag (1996)
19. Grant-Duff, Z.N., Harrison, P.G.: Parallelism via homomorphisms. Parallel Processing Letters 6(2), 279–295 (January 1996)
20. Hamilton, G., Jones, N.: Distillation and labelled transition systems. Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation pp. 15–24 (January 2012)
21. Harel, D.: On folk theorems. Commun. ACM 23, 379–389 (July 1980)
22. Heinz, B.: Lemma discovery by anti unification of regular sorts. Forschungsberichte des Fachbereiches Informatik, Leiter der Fachbibliothek Informatik, Sekretariat FR 5-4 (1994)
23. Hillis, W.D., Steele, Jr., G.L.: Data parallel algorithms. Commun. ACM 29, 1170–1183 (December 1986)
24. Hu, Z., Iwasaki, H., Takechi, M.: Formal derivation of efficient parallel programs by construction of list homomorphisms. ACM Trans. Program. Lang. Syst. 19, 444–461 (May 1997)
25. Hu, Z., Takeichi, M., Chin, W.N.: Parallelization in calculational forms. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 316–328. POPL '98, ACM, New York, NY, USA (1998)
26. Hu, Z., Takeichi, M., Iwasaki, H.: Diffusion: Calculating efficient parallel programs. In: In 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99. pp. 85–94 (1999)
27. Iwasaki, H., Hu, Z.: A new parallel skeleton for general accumulative computations. International Journal of Parallel Programming 32, 389–414 (2004)
28. Keller, G., Chakravarty, M.M.: Flattening trees (1998)
29. Klimov, A.V.: An approach to supercompilation for object-oriented languages: the java supercompiler case study (2008)
30. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. J. Log. Program. 11(3-4), 217–242 (1991)
31. Malcolm, G.: Homomorphisms and promotability. In: Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University. pp. 335–347. Springer-Verlag, London, UK (1989)
32. Matsuzaki, K., Kakehi, K., Iwasaki, H., Hu, Z., Akashi, Y.: A fusion-embedded skeleton library. In: Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference. pp. 644–653. Springer (2004)
33. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. pp. 146–155. PLDI '07, ACM, New York, NY, USA (2007)
34. Nishimura, S., Ohori, A.: A calculus for exploiting data parallelism on recursively defined data. In: In Proc. International Workshop on Theory and Practice on Parallel Programming, LNCS. pp. 413–432. Springer-Verlag (1994)
35. Pettorossi, A., Proietti, M.: Rules and strategies for transforming functional and logic programs (December 1995)
36. Sheard, T., Fegaras, L.: A fold for all seasons. In: Proceedings of the conference on Functional programming languages and computer architecture. pp. 233–242. FPCA '93, ACM, New York, NY, USA (1993)
37. Skillicorn, D.B.: The bird-meertens formalism as a parallel model. In: Software for Parallel Computation, volume 106 of NATO ASI Series F. pp. 120–133. Springer-Verlag (1993)
38. Skillicorn, D.B.: Architecture-independent parallel computation. Computer 23, 38–50 (December 1990)
39. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. ACM COMPUTING SURVEYS 30 (1998)
40. Takano, A., Hu, Z., Takeichi, M.: Program transformation in calculational form (1998)

41. Teo, Y.M., Chin, W.N., Tan, S.H.: Deriving efficient parallel programs for complex recurrences. In: Proceedings of the second international symposium on Parallel symbolic computation. pp. 101–110. PASCO '97, ACM, New York, NY, USA (1997)